Sender/Receiver Interface For Networking

Document #:	P2762R2
Date:	2023-10-12
Project:	Programming Language C++
Audience:	Networking Study Group (SG4) Library Evolution Working Group

Reply-to:	Dietmar Kühl (Bloomberg) <dkuhl@bloomberg.net>

Contents

This document proposes the addition of senders for asynchronous networking operations to the Networking TS and, ultimately, to the C++ Standard. As the `std::execution` proposal isn't landed, yet, this proposal is kept at a high level and is primarily intended to discuss what a potential interface for asynchronous networking operations could look like.

1 Revisions

1.1 Changes for R2

— Added using a socket's context to the discussion on Obtaining the Scheduler.
— Added discussion of the requirement that the scheduler interface needs to be extensible for non-standard operations
— Added discussion of awaitable senders to be more friendly to coroutines.

1.2 Changes for R1

— Added a section with questions to aggregate decisions to poll (and once polled, record the respective outcome).
— Clarified in the motivation why the Networking TS is targeted.
— Added some explanation why getting the scheduler from a receiver isn't entirely the obvious choice (at the end of Obtaining the Scheduler).
— Added a section with discussions to capture topics which were brought up and provide suitable answers and changes.
— Added a section with Network Algorithms [net.algorithms] for resolution.
— Added a section Sender Adaptors to explain why the networking senders should be sender adaptors.
— Enhanced the wording section and added `async_wait`.

2 Motivation

The `std::execution` proposal (P2300) proposes sender/receiver as a general framework for structured and composable concurrency. The currently proposed components define a framework primary targeted at concurrent execution within a program. If this framework gets adopted, it should be possible to integrate other asynchronous work like networking. To facilitate such integration, it is necessary to define a suitable set of senders for the relevant asynchronous network operations.

The proposed changes augment the Networking TS rather than defining entirely new networking components. The main dispute over the Networking TS is the model for asynchronous execution. The vocabulary used to interact with networking facilities like addresses, socket, protocols, etc. doesn't need to be replaced entirely to support sender/receiver. The sender/receiver capabilities can be added which is what the current proposal does.

If the current direction of having only one model for asynchronous operations is confirmed it will need to also remove the executor-based facilities.

3 Related Work

The components defined by P2300 provide a complete framework for managing asynchronous operations and no other facilities beyond senders for the managing the networking operations are needed. The Networking TS defines its own framework for asynchronous operations. This paper does *not* propose the removal of the other framework; whether the asynchronous framework from the Networking TS should be retained or removed is a separate discussion.

There is a proposal for Standard Secure Networking (P2586). The current proposal consists of a high level description and a few possibly usage examples. Based on the usage example, this proposal does *not* include any binding to an asynchronous system. At most, it gets to the question on whether a coroutine interface should be provided to its "poll" facility. So far I haven't created a binding of the interfaces in this proposal to the facilities proposed by P2586 but I don't think there would be any problem doing so. From the current document, it isn't clear to me whether an active "poll" can be interrupted to add new work. Whether it would be a reasonable implementation choice to use P2586 as the base implementation isn't quite clear, as it seems beneficial to potentially use a completion interface, e.g., io_uring, directly.

The focus of P2586 is secured networking and I haven't managed to experiment with a secured version of the proposed networking senders, yet.

P2586 makes some claims about allocations needed for a design based on P2300; there is actually no need to do any allocations at all! The current experimental implementation (it is part of my experimental standard library) doesn't use any allocations in the networking senders (unless variable sized scatter/gather buffers are specified in a way incompatible with an array of `iovec`). When using `poll()` to wait for activity, currently a `std::vector<::pollfd>` and a `std::vector` of completions are used. However, it would be easily possible to specify an interface to a suitable I/O context providing control over the maximum size of these arrays and their required memory to avoid any allocations

4 Design Choices

The basic interface of the senders for the asynchronous network operations is informed by the Networking TS: the available operations and their arguments will be similar. Even so there are some design choices. In most cases, the alternatives aren't exclusive and multiple variations can be supported to support different uses.

The sample code for the different considerations concentrates on the respective choice being considered. As other design choices may affect the resulting code, one of the corresponding options is picked. The different design choices are mostly orthogonal, although some of the choices (notably whether the operations should be member functions) may limit the possibilities for other considerations.

4.1 Obtaining the Scheduler

The networking operations need a suitable context dealing with the asynchronousity, i.e., something using `poll(2)`, `epoll(2)`, `kqueue(2)`, `io_uring(2)`, completion ports, etc., to schedule the operation. The context is abstracted by a scheduler capable of scheduling the respective networking operations. There are a few options for how the scheduler can be obtained:

1. The operation is used as a sender factory and the scheduler is passed in as an argument. This approach makes the scheduling explict when creating the asynchronous operation, e.g.:

```
auto make_accept(auto scheduler, auto& socket) {
    return async::accept(scheduler, socket);
}
```

2. The operation is used as a sender adapter and the scheduler is obtained using `get_completion_scheduler<set_value_t>(s)` from the upstream sender, e.g.:

```
auto make_accept(auto scheduler, auto& socket) {
    return schedule(scheduler)
         | async::accept(socket)
         ;
}
```

3. The operation is used as a sender factory and the sheduler is obtained using `get_scheduler(get_env(r))` from the downstream sender. This approach allows imbuing a work graph with a scheduler specified from the usage end, e.g.:

```
auto make_accept(auto scheduler, auto& socket) {
    return on(scheduler, async::accept(socket));
}
```

4. It was pointed out networking objects like sockets are specific to a context: that is necessary when using I/O Completion Ports or a TLS library and yields advantages when using, e.g., `io_uring(2)`. Explicitly using a separate scheduler for the networking operation could lead to mismatches detected only at run-time. It may, thus, be reasonable to schedule networking entirely based on the objects operation on:

```
auto make_accept(auto& socket) {
    return async::accept(socket);
}
```

The most useful of these options seems to be the third one, i.e., injecting the used scheduler from the point where the asynchronous work is actually used. The other two options require knowledge of the scheduler while building up the asynchronous work.

As a potential variation of the second and third option, a specific customization point name, e.g., `get_completion_io_scheduler` or `get_io_scheduler`, could be used. Using different names could enable the separation of the I/O scheduler from schedulers dedicated to doing work. These queries could fall back to the respective non-specific queries when not provided.

Unfortunately, getting the scheduler from the usage side has an implication on how the work graph is built: for some schedulers, the various operations may need to schedule additional asynchronous operations. For example, a receive operation using TLS may need to set up a way of sending and receiving bytes from a lower level scheduler. When the scheduler is injected through the receiver the operation and used scheduler are brought together rather late, i.e., when `connect(sender, receiver)`ing and when the work graph is already built.

For the examples below, the third option is assumed. However, all three options are viable candidates.

4.2 Error Reporting

For the Networking TS, errors of asynchronous operations are reported using an `std::error_code` argument as part of the completion signature. As there is exactly one completion function used, there isn't really a different alternative. Using receivers supports multiple completion functions, thereby allowing multiple choices:

1. The operation could complete using one `set_value` call using the same fused completion consisting of an `std::error_code` and the other completion arguments as the Networking TS does, e.g.:

```
auto sender
    = async::read(socket, buffer)
    | then([](error_code const& ec, int n) { ... }
    ;
```

   Within a coroutine, structured binding could be used to decompose the result, e.g.:

```
auto[ec, n] = co_await async::read(socket, buffer);
```

2. As the error path is different in the completing functions, it can be reasonable to call different `set_value` functions: one with an `std::error_code` argument and another one (or even multiple ones) with the arguments for the success case. This approch wouldn't work with coroutines as these are restricted to using just one completion signature. Also, a downstream sender would need an overloaded `set_value` to deal with the result:

```
auto sender
    = async::read(socket, buffer)
    | overload(
        [](int n){ /* success path */ },
        [](error_code const& ec){ /* error path */ }
      )
    ;
```

3. Similar to the previous alternative but instead of reporting errors using the `set_value` channel using the `set_error` channel, e.g.:

```
auto sender = async::read(socket, buffer)
    | then([](int n) { /* success path */ }
    | upon_error([](error_code const& ec) { /* error path */ }
    ;
```

   While this approach works with coroutines, it would end up using exceptions, e.g.:

```
try {
    int n = co_await async::read(socket, buffer);
    // success path
} catch (error_code const& ec) {
    // error path
}
```

   However, when using coroutines, it would be possible to use a generic algorithm fusing the `set_value` and the `set_error` results back into one `set_value` result to avoid an exception.

4. There may even be space for a combination of reporting some errors using `set_value` while reporting others using `set_error`, depending on the severity of the error.

5. With senders getting composed in a structured form, it may be reasonable to offer passing a reference to an `std::error_code` and populating that when present and otherwise reporting the error on the `set_error` channel. That would be similar to the synchronous networking operations of the Networking TS reporting errors through the passed argument or an exception. This approach would work reasonably well using coroutines:

```
error_code ec;
int n = co_await async::read(socket, buffer, ec);
if (!ec)
    /* success path */;
else
    /* error path */
```

The most basic variations seems to use a combination of `set_value` for the successful case and `set_error` for the failure cases; the other combinations can be build from that. Also, recognizing an error can be used by algorithms to decide continuing differently upon error, e.g., cancelling other operation for a `when_all`.

However, some of the error cases may have been partial successes. In that case, using the `set_error` channel taking just one argument is somewhat limiting. On the other hand, when substantial work is done and partial

successes become reasonable, it is likely that intermediate results are to be produced and algorithms of a different shape are used anyway.

When using asynchronous operations within a coroutine, there is only one `set_value` supported which can, however, return multiple values using a `std::tuple`, that is then likely decomposed using structured binding. That is when using coroutine defining different `set_value` channels isn't an option. For a coroutine, the `set_error` channel would be turned into an exception. With variations of the asynchronous operations taking an optional `std::error_code` reference as an argument, the coroutine experience would be similar to the synchronous code. Likewise, a coroutine-friendly version of the operations can be provided.

It is possible to offer a combination of the different options. The design choice would name the operations (or the namespace they live in) appropriately. The examples here assume using `set_value` and `set_error` for success and error handling.

4.3 Member vs. Non-Member Operation

The Networking TS uses both member and non-member functions for its operations. Member functions are what users are used to from other languages, where there often aren't different option. The problem is compounded by many IDEs providing simple name completions for member functions. For example:

```
auto sender = socket.async_read(buffer)
    | then([](auto&&...){ /* use result */ })
    ;
```

Similarly, when using coroutines:

```
auto[ec, n] = co_await socket.async_read(buffer);
```

On the hand side, CPOs can't be member functions (well, CPOs are classes with function call *member* functions but they don't really *look* like member functions and they aren't a member of some entity operated on). Also, adding members to classes tends to lead to "kitchen sink" classes acquiring ever more operations over time (see, e.g., `std::basic_string`). The potentially fairly large number of variations (see other design choices) is probably easier managed using non-member function. For example, there may be groups of operations in different namespaces based on their intended use, e.g., `async` for the senders directly used to chain operations and `coro` for senders used within coroutines.

```
auto sender = async::read(socket, buffer)
    | then([](auto&&...){ /* use result */ }
    ;
```

Similarly, when using coroutines:

```
auto[ec, n] = co_await coro::read(socket, buffer);
```

As there is generally no entity used with the P2300 algorithms, these aren't member of classes. For the network operations there is the socket providing an entity and the operations could be defined as member functions of these. Using non-member names providing the full variation of options doesn't exclude using member functions for the expected likely use cases, probably just delegating to the respective non-member operations. The examples here don't use member functions.

4.4 I/O Scheduler Interface

The networking (or, more general, I/O) operations will require being scheduled on a special context and being run on a corresponding scheduler. Also, it is likely desirable to support different schedulers, e.g., one using the most efficient real implementation, one being friendly to integration with other language's "run loop", and one allowing unit testing of networking operations. There are multiple options for how the networking operations are talking to the scheduler:

1. The networking operations and the scheduler use a secret channel. While that is probably the easiest to specify, it means that the networking operations can't accept a somehow adapted scheduler or there needs to be a protocol for how to extract the underlying scheduler.
2. Expose/abstract the various I/O operations somehow, possibly using virtual functions or, more likely, CPOs. While this approach is probably more generic, the interface to the operations is likely at a somewhat lower level than what the senders use and it is possibly platform specific. For example, a `read_some` operation used with `io_uring(2)` needs to provide a pointer to an `iovec` which needs to stay around until the operation is consumed from the completion ring buffer. That is a rather different interface than the generic buffers passed to `read_some`. It may be possible to define the scheduler interface such that it defines what the caller has to store until completion but I haven't tried implementing this approach, yet.
3. The scheduler interface may model multiple contracts (one for each support I/O operation) and each operation produces an object which gets embededded into the I/O operation's operation state object. Each supported underlying I/O interface could store its data in exactly the form needed.
4. The scheduler interface for I/O operations may be what is being proposed by the Low level file i/o library. I haven't tried to implement that.
5. The Networking TS may be doing something in that area and it may be possible to integrate with that or, at least, do something similar. I haven't tried to implement that. It seems the `io_context` uses a secret interface.
6. Aside from networking there are plenty of other events which could be waited for: other forms of I/O, process termination, signals, etc. Ideally, it should be possible to wait for all of these things using just one thread. Not all of the operations are going to be standardized (at least, not initially) and it is necessary to somehow allow applications to expand the support and also wait for additional events.

Most likely, it is preferable to have some form of I/O scheduler abstraction than using a secret interface. However, it isn't yet clear how such an interface would actually look like.

4.5 Timer Class or Just a Sender

The Networking TS defines a `basic_waitable_timer` class template. The type of this class encodes various timer properties like the underlying clock type and some wait traits. The primary need in the Networking TS for this class is the need for an entity to trigger cancellations: while operations are cancellable the cancellation needs to be explicitly wired up where necessary. Uses would look like

```
waitable_timer timer(/* timer settings */);
auto sender = wait_for(timer, 5s);
```

or using coroutines

```
co_await wait_for(timer, 5s);
```

When using sender/receiver cancellation and its necessary wiring is handled by the senders capable of cancelling operations by appropriate use of the receiver's stop token. Correspondingly, there isn't really a need for a timer class. Having to create a timer object and keeping it around is sometimes a bit annoying. Thus, it may be reasonable to allow defining timers simply by creating a suitable sender which is then scheduled on a suitable scheduler. The I/O schedulers are capabable of executing timers.

```
auto sender = wait_for(5s);

co_await wait_for(5s);
```

As with most of the other design choices it may be reasonable to support both alternatives: sometimes it may be reasonable to just schedule a timed operation without the need for an object and specifying the required properties when doing so works. In other situations it may be preferable to encapsulate the timer properties into an object and using this object to schedule multiple timed operations. In that situation it *may* be possible to define the timed operations in a way which doesn't require the timer entity to stay alive until the timed operation completes: removing the need for the timer entity to remain valid until the timed operation completes should make their use simpler. The actual timed operation would be maintained by the scheduler.

4.6 Higher Level Tools

The basic networking or I/O operations are fairly straightforward and there are actually not that many of them (the `io_uring` operations are probably a good indication of the overall scope including operations beyond networking). Concentrating on networking operations the Networking TS doesn't provide everything `io_uring` does. Beyond the basic operations provided by the underlying system, the networking operations can reasonably be composed into higher level algorithms. For example, an `async::read_some` operation potentially reading partial buffers successfully can be composed into an `async::read` operations always reading a complete buffer or failing. The question is, what algorithms should be included in the proposal, if any?

Algorithms like `async::read` and `async::write` are somewhat obvious examples. Something like an `async::resolve` could be an example of a rather non-trivial algorithm: there is the synchronous `getaddrinfo(3)` function, but there doesn't seem to be an asynchronous alternative. With an asynchronous framework in place, it seems reasonable to include an asynchronous version of `getaddrinfo(3)`.

Beyond sender algorithms, there may also be some other interesting components:

— It may be useful to have a coroutine task (`io_task`) injecting a scheduler into asynchronous networking operations used within a coroutine together with a suitable scope (`io_scope`) similar to `async_scope` but also tied to some I/O scheduler. The corresponding task class probably needs to be templatized on the relevant scheduler type.
— For full-duplex operation of a socket, i.e., scheduling concurrent reading and writing operation, something like a ring buffer with sender interfaces to the production and consumption of buffers seems useful.

There are probably various other useful algorithms and components.

4.7 Sender Adaptors

The senders for the networking operations could either be sender factories or sender adapters (see [async.ops] for a definition). Defining the networking operations as sender adaptors has the advantage that customization preferences, e.g., an allocator to be used for variable-sized scatter/gather arguments, can easily be specified. Also, it makes it easier to chain operations. To further facilitate chaining, the networking senders are specified to be pipeable sender adaptors [exec.adapt.objects].

As it is probably quite common that a networking operation starts a work graph they are defined to also accept all argument directly as a convenience. For example, the following senders have the same effect when `started`:

```
auto s0 = execution::just(buffer) | net::async_read_some(socket);
auto s1 = net::async_read_some(execution::just(buffer), socket);
auto s2 = net::async_read_some(socket, buffer);
```

This approach works nicely when the operation takes arguments which can be provided by the upstream sender. Specifically for there is a a problem because `async_accept` only takes the acceptor socket as argument which is expected to be passed directly. Thus, the respective operations would look like this:

```
auto s0 = execution::just() | net::async_accept(acceptor);
auto s1 = net::async_accept(execution::just(), acceptor);
auto s2 = net::async_accept(acceptor);
```

To have both `s0` and `s2` work as senders, the result of `net::async_accept(acceptor)` needs to be both a sender and a sender adaptor closure. However, the specification in [exec.adapt.objects] p2 explicitly prohibits that. There are some options to deal with the case:

1. `async_accept` doesn't create a pipeable sender adaptor and a different approach needs to be used to depend on the completion of prior work, e.g., `s1`.
2. The two calls are distinguished by passing a tag argument to one or both of the alternatives, e.g.:

   ```
   auto s0 = execution::just() | net::async_accept(await_sender, acceptor);
   ```

3. Deal with case special and allow `net::async_accept(acceptor)` to be both a sender and a sender adaptor closure somehow.

4.8 Awaitable Senders

While senders can be turned into awaiters using a promise type's `await_transform`, there are some drawbacks to this approach:

1. When the I/O operation wouldn't block, e.g., because some data could be immediately read or written, it could be detected in `await_ready()` that this is the case. For example, `await_ready()` could attempt whether a non-blocking operation succeeds and, if so, signal that the awaiter is ready. When wrapping a sender, doing the same isn't quite possible.
2. While the standard library can make sure that all coroutines provided are sender-aware, user-defined coroutines would need to explicitly support senders, too. If the networking sender (or possibly even senders in general) were coroutine aware and defined an `operator co_await()` or implemented the awaiter interface, they would be readily usable in any coroutine.

It seems reasonable to specify that the networking senders are awaitables. Having them be awaiter directly would effectively mean that they already include significant state. Making them awaitable would effectively allow creation of an operation state as the result of a call to `operator co_await()` which is a bit tailored to the needs of coroutines. The same may be true for other standard library senders, too.

5 Cancellation Concern

The networking operations are generally inactive after the operation was started but the network operation hasn't completed yet. To cancel such an operation, it is necessary to actively trigger some cancellation function, i.e., a simple test of an atomic `bool` provided by a stop token generally won't work. Thus, the various operations need to register a callback with the receiver's stop token. In cases where the stop token isn't `no_stop_token`, this registration needs to do some synchronization both for the registration and the deregistration of the callback. Repeatedly doing that operation while processing data on a socket may be a performance concern.

It may be possible to avoid setting up cancellation for individual operations and rather hook the cancellation once to a suitable entity like a socket. The corresponding approaches will need some level of support by the library. For example, it may be necessary to support calls to cancel some operations on a socket.

Even when doing so, it may be necessary to inhibit registration of callbacks with a stop token. For example, `when_all` will use receivers using a different stop token than `no_stop_token` with its various senders. In that case it may be useful to have a sender adapter which passes through all completion signals received but always exposes a `no_stop_token` to its sender.

Some systems already have some cancellation support for common use cases. For exmaple, `io_uring(2)` supports timeouts for its operations (using `IORING_OP_LINK_TIMEOUT`). To easily tap into this pattern, it may be reasonable to have a corresponding `timeout` sender taking a time and sender which may be a networking operation: if either the time expires or the sender completes, the respective other operation would be cancelled. Where available, the `timeout` could then just setup the underlying system to provide the corresponding functionality. Otherwise, it would behave like a `timer` and another sender given to a `when_any` operation completing appropriately upon completion of the first operation and cancelling the other via a stop token.

This area still needs some experimentation and, I think, design. The general direction of encapsulating the cancellation into the asynchronous operations is rather interesting, but it isn't a priori clear how to avoid potential costs.

6 Discussion

This section aggregates topics discussed that aren't covered elsewhere in the document. It isn't intended to record the actual discussion we had but rather to cover how to address the respective concerns.

6.1 Support A system execution context

During the SG4 discussion at Issaquah (2023-02-09), support for a system execution context, e.g., based on Grand Central Displatch/libdispatch was discussed. It should be straight forward to expose any of the networking operations using such a context assuming the underlying facilities can support the respective operations. When looking at creating a context using Grand Central Displatch/libdispatch to implement the networking operations, I could only find operations for asynchronous reading and writing but no connection management (accept/connect/shutdown).

The work on the respective facilities should be suitably coordinated with the people proposing a system execution context.

6.2 Resolution

DNS offers name resolution as well as reverse look-up. Name resolution is important as explicit use of IP addresses should be avoided. Name resolution can be implemented as an algorithm using other networking operations.

It was suggested to implement name resolution in terms of libcares. This library is operates in terms of readiness indication of sockets specified via the native handles. Based on the readiness indication it directly uses OS level network functionality to send or receive messages. In that sense, an implementation using libcares isn't an algorithm on top of other networking operations.

An implementation of resolution using other networking operation is probably preferable as the various operations could be customized in some form. For example, to support testing a context implementation could respond with predicatable results replicating various scenarios including error scenarios which may be harder to construct based on a real implementation. However, an implementation using, e.g., libcares should be viable. To support an implementation using libraries requiring a readiness indication, an `async_wait()` operation is added.

The scope of the specified resolution interfaces is limited to simple name and address look-up. For example, DNS support response yielding multiple results. For the initial specification, only look-ups by name and address (reverse look-up) are supported and in both cases only at most one result is considered. More advanced functionality can be added by separate proposals.

Most likely sequence senders using `set_next()` to yield multiple results before completing should be used. However, there is currently (2023-09-01) no proposal for sequence senders.

Implementations should probably be encouraged to use secured DNS approaches where possible. Whether doing so is actually viable still needs to be researched.

6.3 Naming

At the Issaquah SG4 discussion (2023-02-09), it was suggested to drop the `async_` prefix from the names of the networking operations. Instead, the blocking operation should get a prefix like `blocking_` or `sync_` as these operations should be rarely used and the "good" name should be reserved for the commonly used alternative.

The current naming scheme follows the names from the Networking TS. Renaming the operations should be straight forward if that's the decision.

As usual, there are plenty of options how to name things. When considering naming, here are a few alternatives to consider:

1. The current approach is to use the operation with/without prefix.
2. Both the asynchronous and the blocking approach could get a prefix to always clearly indicate what is being used.
3. Instead of using prefixes, the operations could be in different namespace assuming the operations are non-members. This alternative may be especially reasonable if there is also a special variation for the use with coroutines.

4. One set of operations could be member functions while the others are non-member functions. As the customization point approach of sender/receiver lends itself better to non-member functions the non-member use could be asynchronous and the member use could be synchronous.
5. To further the asynchronous approach the standard library could choose not to provide any synchronous interface at all! Instead, users wanting to use a the operations synchronously would use `sync_wait(sender)` also clearly indicating that the operation would be blocking (although this exact alternative probably yields a result too annoying to use and a custom function would be in order).

Ideally, the choice of naming isn't changed every time the proposal is discussed: once a naming scheme is chosen, it should be kept unless new information emerges which proves the current choice unsuitable.

6.4 Constraints On Used Scheduler

The various socket types are created with a specific context. Executing any of the operations on a scheduler not refering to the corresponding context is an error. While the underlying system may not associate a specific context with any socket, the C++ implementation may require additional information for its operation.

In the past the normal approach to similar misuse, e.g., comparing iterators from different containers of the same type, was to treat it as undefined behavior. In the context of networking operation it seems the misuse can be detected at reasonable cost and can be turned into an error, instead. Pursuing this direction the actual error should probably be specified, e.g., in form or a suitable error code and error category combination.

6.5 Required Features

During discussions this proposal various features were mentioned. While this proposal mostly targets adding sender-versions of the operations in the Networking TS (and in its current form it is quite complete at doing so) it seems there are other features also required to make the result an acceptable addition to the C++ standard.

This section provides a brief outline of various features mentioned in preparation to poll how to classify them:

— *required*: without the feature networking shall not be standardized
— *optional*: it would be great to have it but it is viable to standardize networking witout
— *separate*: the feature should be added by a separate proposal

6.5.1 Extensible Scheduler Interface

It was stated that a scheduler interface for networking operations needs to be defined. The Networking TS doesn't do so. Having a defined interface should allow extending the implementation by custom implementations. For example, a test implementation where network operations and their errors are simulated to test applications using it is rather useful for unit tests. Likewise, a logging implementation which behaves like a filter and logs the observed operation at some details would be good. There are probably other uses cases.

The details of such an interface are currently still unclear. Most likely the chosen implementation shouldn't actually affect the types of interfaces as otherwise everything using networking operations would need to be templatized on the scheduler interface. In any case definition of such an interface will require design work.

6.5.2 TLS Support

Secured network communication is rather important. TLS support can be added to the implementation by binding a library like openssl (this is an example and there are other implementations). Especially, if an extensible scheduler interface is defined it is viable to implement secured networking using this facility.

If TLS is to be defined there are different choices. Using, for example, openssl it is fairly straight forward to use a readiness indicator (`async_wait`) to make the operations non-blocking. However, the resulting implementation would directly use an OS-level interface like `recvmsg`/`sendmsg` to interface the network rather than using `async_receive`/`async_send`. In return the result may be more efficient.

It can be argued that binding a library like openssl isn't entirely trivial and shouldn't be left to the user.

6.5.3 Buffer Pools

When having many connections awaiting data it can be wasteful if each connection needs to be provide a read buffer for data to be received. Instead, a buffer pool could be specified, possibly using just one buffer shared between thousands of connections. System interfaces using completion signals like io_uring support special operations such that the buffer is obtained by the kernal from a pool, making it desirable to expose the functionality. The is currently no design for buffer pools.

Of course, if there are many clients the application is probably facing the Internet and a secured connection is used. In that case the state needed to maintain a TLS connection is probably already significantly contributing to the per connection data and sharing the buffer may not make a huge difference.

6.5.4 Async Streams

Defining basic asynchronous networking operations should, in general, allow creation of reasonable networking aware applications. To actually make the creation easy an interface for asychronous streams may be needed. How the interface for such streams would look like is a separate question. To me even the exact objective isn't even clear. There s currently no design for buffer pools.

6.5.5 Networking Algorithms

It was stated that networking algorithms are needed. Hopefully, the available operations allow reasonably easy implementation of relevant algorithms. Of course, not every user should be required to write all relevant algorithms raising the questions: what are the relevant algorithms and which of these should be supported? Here is a probably incomplete list:

- `resolve_name(name)`: resolve `name` into an address/endpoint or a sequence of addresses/endpoints.
- `resolve_address(address)`: obtain a name or a list of names for `address` (reverse resolution).
- `async_read`/`async_write`: like `async_read_some`/ `async_write_some` but process the entire available buffer. There is a bit of design for these algorithms to deal with partial results and progress: maybe using a sequence sender (for which there is no paper, yet) would be reasonable.
- `timeout(duration, sender)`: essentially `when_any(resume_after(duration), sender)` although probably with a nicer interface to consume an upstream sender and produce nicer to use results (although `when_any` isn't proposed by the `std::execution` proposal). Also, in some situations there are more effective ways to implementation a time out.
- `async_connect(socket, endpoints)`: like `async_connect` but use a sequence of endpoints, trying to connect each one in turn until the connection succeeds.
- `async_connect(socket, name)`: like `async_connect(socket, endpoints)` but with the sequence of endpoints obtained from name resolution of `name` rather than a list of readily resolved endpoints.

Which other algorithms are required?

7 Questions

- Should networking support only a sender/receiver model for asynchronous operations, i.e., should the Networking TS's executor model be removed?
- How should the asynchronous networking operation be named? See the Naming section for discussion.
- Should misuse, e.g., executing an operation on a scheduler not matching a socket's context, be an error or undefined behavior?
- What features are necessary (see above for some details)?
  - Extensible scheduler interface
  - TLS support
  - Buffer pools
  - Async streams

— resolve_name
— resolve_address
— async_read/async_write
— timeout
— connect(socket, endpoints)
— connect(socket, name)

8 Wording for Networking CPOs

In 14.2 [io_context.io_context], add a `scheduler_type` and a `get_scheduler()` method to the synopsis:

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
    class io_context : public execution_context
    {
    public:
        ...
        class executor_type;
        class scheduler_type;
        ...
        executor_type  get_executor() noexcept;
        scheduler_type get_scheduler() noexcept;
        ...
    };
}
}
}
}
```

In 14.2 [io_context.io_context] after paragraph 2, add a new paragraph:

2 `count_type` is an implementation-defined unsigned integral type of at least 32 bits.

3 `scheduler_type` is a type modelling *scheduler* [exec.sched].

In 14.2.1a [io_context.io_context.members] after paragraph 3 add a new paragraph:

`executor_type get_executor() noexcept;`

3 *Returns:* An executor that may be used for submitting function objects to the `io_context`.

`scheduler_type get_scheduler() noexcept;`

4 *Returns:* A scheduler that may be used for submitting sender objects to the `io_context`.

Add a new section for the networking operations:

9 Networking Senders [net.sender]

9.1 General [net.sender.general]

1 Subclause [net.sender] defines sender adaptors [async.ops] for networking operations. When the corresponding operations are started, they do not block any thread. Instead they complete once the corresponding operation becomes ready. How the system determines that an operation is ready is implementation specific.

2 These sender adaptors share some common behavior:

(2.1) — The specification for sender adaptors [exec.adapt.general] applies to the sender adaptors in [net.sender]. The sender adaptor in [net.sender] are pipeable sender adaptors [exec.adapt.objects].

(2.2) — When a sender `s` is `connect`ed [exec.connect] to a receiver `r` and the resulting operation state is `execution::start`ed [exec.opstate.start], a callback for cancellation is registered with the stop token obtained using `get_stop_token(get_env(r))`. When this callback is invoked the corresponding operation is cancelled and one of the completion signatures is invoked in a timely manner. The operation can complete using `set_stopped` [exec.set.stopped] but it may still complete with one of the other completion signals instead if the operation became ready otherwise.

(2.3) — Starting an operation state [exec.opstate.start] may complete the operation immediately from the starting thread if it is ready to be completed. Otherwise, the operation is initiated using the scheduler and gets completed once it becomes ready.

(2.4) — Any object referenced in the sender call needs to stay valid while the sender or an operation state obtained from the sender by connecting it to a receiver is used. A sender for a network operation stops being used when it gets connected to a a receiver or when it gets destroyed. An operation state stops being used when it is destroyed or when a completion signal is invoked after the operation state was `execution::start`ed [exec.opstate.start].

(2.5) — The customization points for the sender adaptors also provide an overload taking the arguments which would come from the adapted sender directly. The behavior is as if `execution::just` [exec.just] with the arguments is used as sender argument.

(2.6) — When a sender created by one of the operations taking a socket reference `sock` (however it is named) as argument is `connect`ed to a receiver `r`, it is an error if a scheduler obtained from `get_scheduler(get_env(r))` does not refer to the context used to create `sock`. [*Note:* In the past the specification for inconsistent use typically made the behavior undefined. With the recent push towards removing undefined behavior it seems prudent to avoid undefined behavior for reasonably detectable error. The misuse should be detectable. It may be necessary to specify the actually error, though. – *end note*]

[*Example*: the following senders have the same effect when `connect`ed and `start`ed:

```
auto s0 = execution::just(buffer) | net::async_read_some(s);
auto s1 = net::async_read_some(execution::just(buffer), s);
auto s2 = net::async_read_some(s, buffer);
```

–*End Example*].

9.2 Network Sender Operations Synopsis [net.sender.syn]

```
namespace std::experimental::net::inline v1 {
    namespace sender-adaptors // exposition only {
        struct async_accept_t;          // [net.sender.async.accept]
        struct async_connect_t;         // [net.sender.async.connect]
        struct async_read_some_t;       // [net.sender.async.read.some]
        struct async_read_t;            // [net.sender.async.read]
        struct async_receive_from_t;    // [net.sender.async.receive.from]
        struct async_receive_t;         // [net.sender.async.receive]
        struct async_resolve_address_t; // [net.sender.async.resolve.address]
        struct async_resolve_name_t;    // [net.sender.async.resolve.name]
        struct async_resume_after_t;    // [net.sender.async.resume.after]
        struct async_resume_at_t;       // [net.sender.async.resume.at]
        struct async_send_t;            // [net.sender.async.send]
        struct async_send_to_t;         // [net.sender.async.send.to]
        struct async_wait_t;            // [net.sender.async.wait]
        struct async_write_some_t;      // [net.sender.async.write.some]
```

```
        struct async_write_t;           // [net.sender.async.write]
        struct timeout_t;               // [net.sender.async.timeout]
        struct when_any_t;              // [net.sender.async.when.any]
    }

    using sender-adaptors::async_accept_t;
    using sender-adaptors::async_connect_t;
    using sender-adaptors::async_read_some_t;
    using sender-adaptors::async_read_t;
    using sender-adaptors::async_receive_from_t;
    using sender-adaptors::async_receive_t;
    using sender-adaptors::async_resolve_address_t;
    using sender-adaptors::async_resolve_name_t;
    using sender-adaptors::async_resume_after_t;
    using sender-adaptors::async_resume_at_t;
    using sender-adaptors::async_send_t;
    using sender-adaptors::async_send_to_t;
    using sender-adaptors::async_wait_t;
    using sender-adaptors::async_write_some_t;
    using sender-adaptors::async_write_t;
    using sender-adaptors::timeout_t;
    using sender-adaptors::when_any_t;

    inline constexpr async_accept_t          async_accept{};
    inline constexpr async_connect_t         async_connect{};
    inline constexpr async_read_some_t       async_read_some{};
    inline constexpr async_read_t            async_read{};
    inline constexpr async_receive_from_t    async_receive_from{};
    inline constexpr async_receive_t         async_receive{};
    inline constexpr async_resolve_address_t async_resolve_address{};
    inline constexpr async_resolve_name_t    async_resolve_name{};
    inline constexpr async_resume_after_t    async_resume_after{};
    inline constexpr async_resume_at_t       async_resume_at{};
    inline constexpr async_send_t            async_send{};
    inline constexpr async_send_to_t         async_send_to{};
    inline constexpr async_wait_t            async_wait{};
    inline constexpr async_write_some_t      async_write_some{};
    inline constexpr async_write_t           async_write{};
    inline constexpr timeout_t               timeout{};
    inline constexpr when_any_t              when_any{};
}
```

9.3 Network Sender Operations [net.sender.operations]

[*Note*: The shape of the exact operations isn't quite clear yet, as there are various design option (see above). Below the relevant operations are listed together with their arguments and their likely `completion_signatures`. The current list of operations may be incomplete. The "as if" code isn't necessarily ready to compile and may omit necessary casts and use private operations in some cases. – *end note*]

9.3.1 `net::async_accept` [net.sender.async.accept]

```
namespace std::experimental::net::inline v1 {
  namespace sender-adaptors // exposition only {
```

```
    struct async_accept_t {
      struct accept-sender; // exposition only

      template <class AcceptableProtocol>
      accept-sender operator()(basic_socket_acceptor<AcceptableProtocol>& acceptor) const;

      template <execution::sender_of<execution::set_value_t()> Sender,
                class AcceptableProtocol>
      accept-sender operator()(Sender&& sender,
                               basic_socket_acceptor<AcceptableProtocol>& acceptor) const;
    };
  }
}
```

[1] `async_accept_t` is the type of customization point objects for creating senders for accepting new socket connections.

```
template <class AcceptableProtocol>
accept-sender
async_accept_t::operator()(basic_socket_acceptor<AcceptableProtocol>& acceptor) const
```

[2] *Returns*: `(*this)(execution::just(), acceptor);`

```
template <execution::sender_of<execution::set_value_t()> Sender,
          class AcceptableProtocol>
accept-sender
async_accept_t::operator()(Sender&& sender,
                           basic_socket_acceptor<AcceptableProtocol>& acceptor) const;
```

[3] The operator creates an *accept-sender*. After the returned *accept-sender* is `connect()`ed it uses the `set_value` completion of `sender` to trigger the start of its own operation. If `sender` completes with a `set_error` or `set_stopped` completion, the completion is forwarded to the connected receiver.

[4] Let `s` be the *accept-sender* returned from `async_accept_t()(sender, acceptor)`. Let `Acceptor` be the type `basic_socket_acceptor<AcceptableProtocol>` and `env` be an environment object. The completion signatures returned from `execution::get_completion_signatures(s, env)` contain the following elements:

(4.1) — `execution::set_value_t(typename Acceptor::socket_type, typename Acceptor::endpoint_type)`

(4.2) — `execution::set_error_t(error_code)`

(4.3) — `execution::set_stopped_t()`

(4.4) — Any additional `execution::set_error_t` signature from `execution::get_completion_signatures(sender, env)`

[5] When `s` is connected to a receiver `r` and the resulting operation state is started, it initiates an asynchronous operation to extract a socket from the queue of pending connections for `acceptor` when there is at least one pending connection, as if by POSIX:

```
typename Acceptor::endpoint_type ep;
socklen_t        addrlen(ep.capacity());
auto h = accept(acceptor.native_handle(), ep.data(), &addrlen);
if (h < 0) {
    execution::set_error(std::move(r), error_code(errno, system_category()));
}
else {
    ep.resize(addrlen);
    execution::set_value(std::move(r), typename Acceptor::socket_type(h), ep);
}
```

[6] [*Note*: The Networking TS passes the `endpoint` optionally as a reference argument, resulting in two interfaces:

one where the endpoint isn't obtained and one where it is. The interface for `async_accept` could reference the `endpoint` as well, instead of providing it with `set_value`. An alternative design allowing omission of the `endpoint` is to have `async_accept` not providing an `endpoint` with `set_value` and `async_accept_from` with the completion signature above. _–End Note___]

9.3.2 `net::async_connect` [net.sender.async.connect]

```
namespace std::experimental::net::inline v1 {
  namespace sender-adaptors // exposition only {
    struct async_connect_t {
      struct connect-sender;                 // exposition only
      struct connect-sender-adaptor-closure; // exposition only

      template <class Protocol>
      connect-sender operator()(basic_stream_socket<Protocol>& socket,
                                const typename basic_stream_socket<Protocol>::endpoint_type& ep) const;

      template <class Protocol>
      connect-sender-adaptor-closure
      operator()(basic_stream_socket<Protocol>& socket) const;

      template <class Protocol,
                execution::sender_of<typename basic_stream_socket<Protocol>::endpoint_type> Sender>
      connect-sender operator()(Sender&& sender,
                                basic_stream_socket<Protocol>& socket) const;
    };
  }
}
```

1 `async_connect_t` is the type of customization point objects for creating senders connecting a stream socket to a peer.

```
template <class Protocol>
connect-sender
async_connect_t::operator()(basic_stream_socket<Protocol>& socket,
                            typename basic_stream_socket<Protocol>::endpoint_type ep) const;
```

2 *Returns*: `(*this)(execution::just(ep), socket);`

```
template <class Protocol>
connect-sender-adaptor-closure
async_connect_t::operator()(basic_stream_socket<Protocol>& socket) const;
```

3 *Returns*: an object `closure` such that `sender | closure` yields an object equivalent to `(*this)(sender, socket)`.

```
template <execution::sender Sender, class Protocol>
connect-sender
async_connect_t::operator()(Sender&& sender,
                            basic_stream_socket<Protocol>& socket) const;
```

4 The operator creates a *connect-sender*. After the returned *connect-sender* is `connect()`ed it uses the `set_value` completion of `sender` to trigger the start of its own operation. It uses the `endpoint_type` argument to `set_value` as the address to connect to. If `sender` completes with a `set_error` or `set_stopped` completion, the completion is forwarded to the connected receiver.

5 Let `s` be the *connect-sender* returned from `async_connect_t()(sender, socket)` and `env` be an environment object. The completion signatures returned from `execution::get_completion_signatures(s, env)` contain

the following elements:

(5.1) — `execution::set_value_t()`
(5.2) — `execution::set_error_t(error_code)`
(5.3) — `execution::set_stopped_t()`
(5.4) — Any additional `execution::set_error_t` signature from `execution::get_completion_signatures(sender, env)`

6 When `s` is connected to a receiver `r` and the resulting operation state is started by a call to `set_value` with argument `ep`, it initiates an asynchronous operation to connect to a peer, as if by POSIX:

```
if (connect(socket.native_handle(), ep.data(), ep.size()) < 0) {
    execution::set_error(std::move(r), error_code(errno, system_category()));
}
else {
    execution::set_value(std::move(r));
}
```

9.3.3 `net::async_read_some` [net.sender.async.read.some]

```
namespace std::experimental::net::inline v1 {
  namespace sender-adaptors // exposition only {
    struct async_read_some_t {
      struct read-some-sender;                  // exposition only
      struct read-some-sender-adaptor-closure; // exposition only

      template <class Protocol, class MutableBufferSequence>
      read-some-sender operator()(basic_stream_socket<Protocol>& socket,
                                  const MutableBufferSequence& buffers) const;

      template <class Protocol>
      read-some-sender-adaptor-closure
      operator()(basic_stream_socket<Protocol>& socket) const;

      template <execution::sender Sender, class Protocol>
      read-some-sender operator()(Sender&& sender,
                                  basic_stream_socket<Protocol>& socket) const;
    };
  }
}
```

1 `async_read_some_t` is the type of customization point objects for creating senders reading a sequence of buffers from a stream socket.

```
template <class Protocol, class MutableBufferSequence>
read-some-sender
async_read_some_t::operator()(basic_stream_socket<Protocol>& socket,
                              const MutableBufferSequence& buffers) const;
```

2 *Returns*: `(*this)(execution::just(buffers), socket);`

```
template <class Protocol>
read-some-sender-adaptor-closure
async_read_some_t::operator()(basic_stream_socket<Protocol>& socket) const;
```

3 *Returns*: an object `closure` such that `sender | closure` yields an object equivalent to `(*this)(sender, socket)`.

```
template <execution::sender Sender, class Protocol>
read-some-sender
```

```
async_read_some_t::operator()(Sender&& sender,
                              basic_stream_socket<Protocol>& socket) const;
```

4 *Returns*: `async_receive(sender, socket);`

5 [*Note*: The customization point may provide additional overloads, e.g., for files. As a result this name may be useful for algorithms applicable to other streams then just sockets in the future. _ – end note_]

9.3.4 `net::async_receive` [net.sender.async.receive]

```
namespace std::experimental::net::inline v1 {
  namespace sender-adaptors // exposition only {
    struct async_receive_t {
      struct receive-sender;                 // exposition only
      struct receive-sender-adaptor-closure; // exposition only

      template <class Protocol, class MutableBufferSequence>
      receive-sender operator()(basic_socket<Protocol>& socket,
                                const MutableBufferSequence& buffers) const;
      template <class Protocol, class MutableBufferSequence>
      receive-sender operator()(basic_socket<Protocol>& socket,
                                sender_base::message_flags flags,
                                const MutableBufferSequence& buffers) const;

      template <class Protocol>
      receive-sender-adaptor-closure
      operator()(basic_socket<Protocol>& socket) const;

      template <class Protocol, execution::sender Sender>
      receive-sender operator()(Sender&& sender,
                                basic_socket<Protocol>& socket) const;
    };
  }
}
```

1 `async_receive_t` is the type of customization point objects for creating senders receiving messages on a connected socket.

```
template <class Protocol, class MutableBufferSequence>
receive-sender
async_receive_t::operator()(basic_socket<Protocol>& socket,
                            const MutableBufferSequence& buffers) const;
```

2 *Returns*: `(*this)(execution::just(buffers), socket);`

```
template <class Protocol, class MutableBufferSequence>
receive-sender
async_receive_t::operator()(basic_socket<Protocol>& socket,
                            sender_base::message_flags flags,
                            const MutableBufferSequence& buffers) const;
```

3 *Returns*: `(*this)(execution::just(flags, buffers), socket);`

```
template <class Protocol>
receive-sender-adaptor-closure
async_receive_t::perator()(basic_socket<Protocol>& socket) const;
```

4 *Returns*: an object `closure` such that `sender | closure` yields an object equivalent to `(*this)(sender, socket)`.

```
template <class Protocol, execution::sender Sender>
receive-sender
async_receive_t::operator()(Sender&& sender,
                            basic_socket<Protocol>& socket) const;
```

5 The operator creates a *receive-sender*. After the returned *receive-sender* is `connect()`ed it uses the `set_value` completion of `sender` to trigger the start of its own operation. It uses the `flags` and `buffers` arguments to `set_value` to specify how to receive data. If the `sender_base::message_flags` parameter is not present it uses a `flags` variable initialized to `socket_base::message_flags()`. If `sender` completes with a `set_error` or `set_stopped` completion, the completion is forwarded to the connected receiver.

6 Let `s` be the *receive-sender* returned from `async_receive_t()(sender, socket)` and `env` be an environment object. The completion signatures returned from `execution::get_completion_signatures(s, env)` contain the following elements:

(5.1) — `execution::set_value_t()`
(5.2) — `execution::set_error_t(error_code)`
(5.3) — `execution::set_stopped_t()`
(5.4) — Any additional `execution::set_error_t` signature from `execution::get_completion_signatures(sender, env)`

6 When `s` is connected to a receiver `r` and the resulting operation state is started by a call to `set_value` with argument `buffers` or arguments `flags` and `buffers`, it initiates an asynchronous operation to receive data. The operation constructs and array `iov` of POSIX `struct iovec` of size `length` corresponding to `buffers` and reads data as if by POSIX:

```
msghdr message;
message.msg_name        = nullptr;
message.msg_namelen     = 0;
message.msg_iov         = iov;
message.msg_iovlen      = length;
message.msg_control     = nullptr;
message.msg_controllen  = 0;
message.msg_flags       = 0;
auto n = recvmsg(socket.native_handle(), &message, static_cast<int>(flags));

if (n < 0) {
    execution::set_error(std::move(r), error_code(errno, system_category()));
}
else {
    execution::set_value(std::move(r), n);
}
```

9.3.5 `net::async_receive_from` [net.sender.async.receive.from]

```
namespace std::experimental::net::inline v1 {
  namespace sender-adaptors // exposition only {
    struct async_receive_from_t {
      struct receive-from-sender;                 // exposition only
      struct receive-from-sender-adaptor-closure; // exposition only

      template <class Protocol, class MutableBufferSequence>
      receive-from-sender operator()(basic_datagram_socket<Protocol>& socket,
                                     const MutableBufferSequence& buffers,
                                     typename basic_datagram_socket<Protocol>::endpoint_type& ep) const;
      template <class Protocol, class MutableBufferSequence>
```

```
      receive-from-sender operator()(basic_datagram_socket<Protocol>& socket,
                                     sender_base::message_flags flags,
                                     const MutableBufferSequence& buffers,
                                     typename basic_datagram_socket<Protocol>::endpoint_type& ep) const;

      template <class Protocol>
      receive-from-sender-adaptor-closure
      operator()(basic_datagram_socket<Protocol>& socket) const;

      template <class Protocol, execution::sender Sender>
      receive-from-sender operator()(Sender&& sender,
                                     basic_datagram_socket<Protocol>& socket) const;
    };
  }
}
```

[1] `async_receive_from_t` is the type of customization point objects for creating senders receiving messages on a socket from a specified endpoint.

```
template <class Protocol, class MutableBufferSequence>
receive-from-sender
async_receive_from_t::operator()(basic_datagram_socket<Protocol>& socket,
                                 const MutableBufferSequence& buffers,
                                 typename basic_datagram_socket<Protocol>::endpoint_type& ep) const;
```

[2] *Returns*: `(*this)(execution::just(buffers, ep), socket);`

```
template <class Protocol, class MutableBufferSequence>
receive-from-sender
async_receive_from_t::operator()(basic_datagram_socket<Protocol>& socket,
                                 sender_base::message_flags flags,
                                 const MutableBufferSequence& buffers,
                                 typename basic_datagram_socket<Protocol>::endpoint_type& ep) const;
```

[3] *Returns*: `(*this)(execution::just(flags, buffers, ep), socket);`

```
template <class Protocol>
receive-from-sender-adaptor-closure
async_receive_from_t::operator()(basic_datagram_socket<Protocol>& socket) const;
```

[4] *Returns*: an object `closure` such that `sender | closure` yields an object equivalent to `(*this)(sender, socket)`.

```
template <class Protocol, execution::sender Sender>
receive-from-sender
async_receive_from_t::operator()(Sender&& sender,
                                 basic_datagram_socket<Protocol>& socket) const;
```

[5] The operator creates a *receive-from-sender*. After the returned *receive-from-sender* is `connect()`ed it uses the `set_value` completion of `sender` to trigger the start of its own operation. It uses the `flags`, `buffers`, and `ep` arguments to `set_value` to specify how to receive data. If the `sender_base::message_flags` parameter is not present it uses a `flags` variable initialized to `socket_base::message_flags()`. If `sender` completes with a `set_error` or `set_stopped` completion, the completion is forwarded to the connected receiver.

[6] Let `s` be the *receive-from-sender* returned from `async_receive_from_t()(sender, socket)` and `env` be an environment object. The completion signatures returned from `execution::get_completion_signatures(s, env)` contain the following elements:

(5.1) — `execution::set_value_t()`

(5.2) — `execution::set_error_t(error_code)`

(5.3) — `execution::set_stopped_t()`

(5.4) — Any additional `execution::set_error_t` signature from `execution::get_completion_signatures(sender, env)`

6 When `s` is connected to a receiver `r` and the resulting operation state is started by a call to `set_value` with argument `buffers` and `ep` or arguments `flags`, `buffers`, and ep, it initiates an asynchronous operation to receive data. The operation constructs and array `iov` of POSIX `struct iovec` of size `length` corresponding to `buffers` and receives data as if by POSIX:

```
typename Socket::endpoint_type addr;
socklen_t                      addrlen{ep.capacity()};
msghdr message;
message.msg_name        = addr.data();
message.msg_namelen     = &addrlen;
message.msg_iov         = iov;
message.msg_iovlen      = length;
message.msg_control     = nullptr;
message.msg_controllen  = 0;
message.msg_flags       = 0;
auto n = recvmsg(socket.native_handle(), &message, static_cast<int>(flags));

if (n < 0) {
    execution::set_error(std::move(r), error_code(errno, system_category()));
}
else {
    addr.resize(addrlen);
    execution::set_value(std::move(r), n, addr);
}
```

9.3.6 `net::async_send` [net.sender.async.send]

```
namespace std::experimental::net::inline v1 {
  namespace sender-adaptors // exposition only {
    struct async_send_t {
      struct send-sender;                 // exposition only
      struct send-sender-adaptor-closure; // exposition only

      template <class Protocol, class ConstantBufferSequence>
      send-sender operator()(basic_socket<Protocol>& socket,
                             const ConstantBufferSequence& buffers) const;
      template <class Protocol, class ConstantBufferSequence>
      send-sender operator()(basic_socket<Protocol>& socket,
                             sender_base::message_flags flags,
                             const ConstantBufferSequence& buffers) const;

      template <class Protocol>
      send-sender-adaptor-closure
      operator()(basic_socket<Protocol>& socket) const;

      template <class Protocol, execution::sender Sender>
      send-sender operator()(Sender&& sender,
                             basic_socket<Protocol>& socket) const;
    };
  }
}
```

[1] `async_send_t` is the type of customization point objects for creating senders sending messages on a connected socket.

```
template <class Protocol, class ConstantBufferSequence>
send-sender
async_send_t::operator()(basic_socket<Protocol>& socket,
                         const ConstantBufferSequence& buffers) const;
```

[2] *Returns*: `(*this)(execution::just(buffers), socket);`

```
template <class Protocol, class ConstantBufferSequence>
send-sender
async_send_t::operator()(basic_socket<Protocol>& socket,
                         sender_base::message_flags flags,
                         const ConstantBufferSequence& buffers) const;
```

[3] *Returns*: `(*this)(execution::just(flags, buffers), socket);`

```
template <class Protocol>
send-sender-adaptor-closure
async_send_t::perator()(basic_socket<Protocol>& socket) const;
```

[4] *Returns*: an object `closure` such that `sender | closure` yields an object equivalent to `(*this)(sender, socket)`.

```
template <class Protocol, execution::sender Sender>
send-sender
async_send_t::operator()(Sender&& sender,
                         basic_socket<Protocol>& socket) const;
```

[5] The operator creates a *send-sender*. After the returned *send-sender* is `connect()`ed it uses the `set_value` completion of `sender` to trigger the start of its own operation. It uses the `flags` and `buffers` arguments to `set_value` to specify how to send data. If the `sender_base::message_flags` parameter is not present it uses a `flags` variable initialized to `socket_base::message_flags()`. If `sender` completes with a `set_error` or `set_stopped` completion, the completion is forwarded to the connected receiver.

[6] Let `s` be the *send-sender* returned from `async_send_t()(sender, socket)` and `env` be an environment object. The completion signatures returned from `execution::get_completion_signatures(s, env)` contain the following elements:

(5.1) — `execution::set_value_t()`

(5.2) — `execution::set_error_t(error_code)`

(5.3) — `execution::set_stopped_t()`

(5.4) — Any additional `execution::set_error_t` signature from `execution::get_completion_signatures(sender, env)`

[6] When `s` is connected to a receiver `r` and the resulting operation state is started by a call to `set_value` with argument `buffers` or arguments `flags` and `buffers`, it initiates an asynchronous operation to send data. The operation constructs and array `iov` of POSIX `struct iovec` of size `length` corresponding to `buffers` and sends data as if by POSIX:

```
msghdr message;
message.msg_name        = nullptr;
message.msg_namelen     = 0;
message.msg_iov         = iov;
message.msg_iovlen      = length;
message.msg_control     = nullptr;
message.msg_controllen  = 0;
message.msg_flags       = 0;
auto n = sendmsg(socket.native_handle(), &message, static_cast<int>(flags));
```

```
if (n < 0) {
    execution::set_error(std::move(r), error_code(errno, system_category()));
}
else {
    execution::set_value(std::move(r), n);
}
```

9.3.7 `net::async_send_to` [net.sender.async.send.to]

```
namespace std::experimental::net::inline v1 {
  namespace sender-adaptors // exposition only {
    struct async_send_to_t {
      struct send-to-sender;                 // exposition only
      struct send-to-sender-adaptor-closure; // exposition only

      template <class Protocol, class MutableBufferSequence>
      send-to-sender operator()(basic_datagram_socket<Protocol>& socket,
                                const MutableBufferSequence& buffers,
                                typename basic_datagram_socket<Protocol>::endpoint_type& ep) const;
      template <class Protocol, class MutableBufferSequence>
      send-to-sender operator()(basic_datagram_socket<Protocol>& socket,
                                sender_base::message_flags flags,
                                const MutableBufferSequence& buffers,
                                typename basic_datagram_socket<Protocol>::endpoint_type& ep) const;

      template <class Protocol>
      send-to-sender-adaptor-closure
      operator()(basic_datagram_socket<Protocol>& socket) const;

      template <class Protocol, execution::sender Sender>
      send-to-sender operator()(Sender&& sender,
                                basic_datagram_socket<Protocol>& socket) const;
    };
  }
}
```

1 `async_send_to_t` is the type of customization point objects for creating senders sending messages on a socket to a specified endpoint.

```
template <class Protocol, class MutableBufferSequence>
send-to-sender
async_send_to_t::operator()(basic_datagram_socket<Protocol>& socket,
                            const MutableBufferSequence& buffers,
                            typename basic_datagram_socket<Protocol>::endpoint_type& ep) const;
```

2 *Returns*: `(*this)(execution::just(buffers, ep), socket);`

```
template <class Protocol, class MutableBufferSequence>
send-to-sender
async_send_to_t::operator()(basic_datagram_socket<Protocol>& socket,
                            sender_base::message_flags flags,
                            const MutableBufferSequence& buffers,
                            typename basic_datagram_socket<Protocol>::endpoint_type& ep) const;
```

3 *Returns*: `(*this)(execution::just(flags, buffers, ep), socket);`

```
template <class Protocol>
send-to-sender-adaptor-closure
async_send_to_t::operator()(basic_datagram_socket<Protocol>& socket) const;
```

4 *Returns*: an object `closure` such that `sender | closure` yields an object equivalent to `(*this)(sender, socket)`.

```
template <class Protocol, execution::sender Sender>
send-to-sender
async_send_to_t::operator()(Sender&& sender,
                            basic_datagram_socket<Protocol>& socket) const;
```

5 The operator creates a *send-to-sender*. After the returned *send-to-sender* is `connect()`ed it uses the `set_value` completion of `sender` to trigger the start of its own operation. It uses the `flags`, `buffers`, and `ep` arguments to `set_value` to specify how to send data. If the `sender_base::message_flags` parameter is not present it uses a `flags` variable initialized to `socket_base::message_flags()`. If `sender` completes with a `set_error` or `set_stopped` completion, the completion is forwarded to the connected receiver.

6 Let `s` be the *send-to-sender* returned from `async_send_to_t()(sender, socket)` and `env` be an environment object. The completion signatures returned from `execution::get_completion_signatures(s, env)` contain the following elements:

(5.1) — `execution::set_value_t()`
(5.2) — `execution::set_error_t(error_code)`
(5.3) — `execution::set_stopped_t()`
(5.4) — Any additional `execution::set_error_t` signature from `execution::get_completion_signatures(sender, env)`

6 When `s` is connected to a receiver `r` and the resulting operation state is started by a call to `set_value` with argument `buffers` and `ep` or arguments `flags`, `buffers`, and ep, it initiates an asynchronous operation to send data. The operation constructs and array `iov` of POSIX `struct iovec` of size `length` corresponding to `buffers` and receives data as if by POSIX:

```
msghdr message;
message.msg_name        = ep.data();
message.msg_namelen     = ep.size();
message.msg_iov         = iov;
message.msg_iovlen      = length;
message.msg_control     = nullptr;
message.msg_controllen  = 0;
message.msg_flags       = 0;
auto n = sendmsg(socket.native_handle(), &message, static_cast<int>(flags));

if (n < 0) {
    execution::set_error(std::move(r), error_code(errno, system_category()));
}
else {
    execution::set_value(std::move(r), n);
}
```

9.3.8 `net::async_wait` [net.sender.async.wait]

```
namespace std::experimental::net::inline v1 {
  namespace sender-adaptors // exposition only {
    struct async_wait_t {
      struct wait-sender;                 // exposition only
      struct wait-sender-adaptor-closure; // exposition only
```

```
      template <class Protocol>
      wait-sender operator()(basic_socket<Protocol>& socket,
                             socket_base::wait_type events) const;

      template <class Protocol>
      wait-sender-adaptor-closure operator()(basic_socket<Protocol>& socket) const;

      template <execution::sender_of<execution::set_value_t(socket_base::wait_type)> Sender,
                class Protocol>
      wait-sender operator()(Sender&& sender,
                             basic_socket<Protocol>& socket) const;
    };
  }
}
```

1 `async_wait_t` is the type of customization point objects for creating senders awaiting readiness of sockets for a specific set of operations. [*Note*: This operation could be seen as the basis operation for readiness-based contexts as all the operations can be implemented in terms of this operation and conceptually blocking calls which are known not to block due to the readiness indicator. – *end note*]

```
template <class Protocol>
wait-sender
async_wait_t::operator()(basic_socket<Protocol>& socket,
                         socket_base::wait_type events) const
```

2 *Returns*: `(*this)(execution::just(events), socket);`

```
template <class Protocol>
wait-sender-adaptor-closure
async_wait_t::operator()(basic_socket<Protocol>& socket) const;
```

3 *Returns*: an object `closure` such that `sender | closure` yields an object equivalent to `(*this)(sender, socket)`.

```
template <execution::sender_of<execution::set_value_t(socket_base::wait_type)> Sender,
          class Protocol>
wait-sender
async_wait_t::operator()(Sender&& sender,
                         basic_socket<Protocol>& socket) const;
```

4 The operator creates a *wait-sender*. After the returned *wait-sender* is `connect()`ed it uses the `set_value` completion of `sender` to trigger the start of its own operation. It uses the `socket_base::wait_type` argument to `set_value` as a bitmask to await readiness of the respective conditions. If `sender` completes with a `set_error` or `set_stopped` completion, the completion is forwarded to the connected receiver.

5 Let `s` be the *wait-sender* returned from `async_wait_t()(sender, socket)` and `env` be an environment object. The completion signatures returned from `execution::get_completion_signatures(s, env)` contain the following elements:

(5.1) — `execution::set_value_t(socket_base::wait_type)`

(5.2) — `execution::set_error_t(error_code)`

(5.3) — `execution::set_stopped_t()`

(5.4) — Any additional `execution::set_error_t` signature from `execution::get_completion_signatures(sender, env)`

6 When `s` is connected to a receiver `r` and the resulting operation state is started by a call to `set_value` with argument `events`, it initiates an asynchronous operation to await readiness of I/O operations on `s`, as if by POSIX:

```
pollfds fds[1];
fds.fd     = s.native_handle();
fds.events = (events & s.wait_read? POLLIN: 0) | (events & s.wait_write? POLLOUT: 0);
if (1 == poll(fds, 1, 0)) {
    execution::set_value(std::move(r),
                         (fds[0].revents & POLLIN? s.wait_read(): socket_base::wait_type()) |
                         (fds[0].revents & POLLOUT? s.wait_write(): socket_base::wait_type()) |
                         (fds[0].revents & POLLERR? s.wait_error(): socket_base::wait_type()));
}
else {
    execution::set_error(std::move(r), error_code(errno, system_category()));
}
```

9.3.9 `net::async_write_some` [net.sender.async.write.some]

```
namespace std::experimental::net::inline v1 {
  namespace sender-adaptors // exposition only {
    struct async_write_some_t {
      struct write-some-sender;                 // exposition only
      struct write-some-sender-adaptor-closure; // exposition only

      template <class Protocol, class ConstBufferSequence>
      write-some-sender operator()(basic_stream_socket<Protocol>& socket,
                                   const ConstBufferSequence& buffers) const;

      template <class Protocol>
      write-some-sender-adaptor-closure
      operator()(basic_stream_socket<Protocol>& socket) const;

      template <execution::sender Sender, class Protocol>
      write-some-sender operator()(Sender&& sender,
                                   basic_stream_socket<Protocol>& socket) const;
    };
  }
}
```

[1] `async_write_some_t` is the type of customization point objects for creating senders writing a sequence of buffers to a stream socket.

```
template <class Protocol, class ConstBufferSequence>
write-some-sender
async_write_some_t::operator()(basic_stream_socket<Protocol>& socket,
                               const ConstBufferSequence& buffers) const;
```

[2] *Returns*: `(*this)(execution::just(buffers), socket);`

```
template <class Protocol>
write-some-sender-adaptor-closure
async_write_some_t::operator()(basic_stream_socket<Protocol>& socket) const;
```

[3] *Returns*: an object `closure` such that `sender | closure` yields an object equivalent to `(*this)(sender, socket)`.

```
template <execution::sender Sender, class Protocol>
write-some-sender
async_write_some_t_operator()(Sender&& sender,
                              basic_stream_socket<Protocol>& socket) const;
```

4 *Returns*: `async_send(sender, socket);`

5 [*Note*: The customization point may provide additional overloads, e.g., for files. As a result this name may be useful for algorithms applicable to other streams then just sockets in the future. _ – end note_]

9.4 Network Algorithms [net.algorithms]

9.4.1 General [net.algorithms.general]

1 The section [net.algorithms] specifies algorithms relevant for networking programming. Whether the corresponding operations are, indeed, implemented as algorithms using lower level networking operations or something different is left unspecified altough implementations are encouraged to do so.

9.4.2 `net::async_resolve_name` [net.sender.async.resolve.name]

1 `async_resolve_name` is a customization point for resolving a name to an address.

2 `async_resolve_name` is parameterized on an `InternetProtocol` and takes a sequence of characters specifying a name to be looked-up as argument. Upon success it returns a `net::ip::basic_endpoint<InternetProtocol>` with a result from the look-up.

3 The completion signatures returned from from `execution::get_completion_signatures(s, env)` contain three elements where `s` is a sender object returned from `net::async_resolve_name<InternetProtocol>`:

(3.1) — `execution::set_value(net::ip::basic_endpoint<InternetProtocol>)`
(3.2) — `execution::set_error_t(error_code)`
(3.3) — `execution::set_stopped_t()`

4 The default implementation uses DNS (as specified by a sequence of RFCs) to resolve the name into an addresses. How the operation is done exactly is implementation specific.

5 [*Editing note:* There should probably a version of the operation yielding a sequence of results. Most likely the proper approach to do is to use sequence senders which are, however, not yet in a any proposal being discussed. – *end note*]

9.4.3 `net::async_resolve_address` [net.sender.async.resolve.address]

1 `async_resolve_address` is a customization point for resolving an address to a name.

2 `async_resolve_name` takes an `net::ip::basic_endpoint<InternetProtocol>` as argument. Upon success it returns a `string_view` with a name for the endpoint. [*Note:* The referenced sequence of characters is only valid until `set_value` returns – *end note*]

3 Let `s` be a sender object returned from `async_reolve_name`. The completion signatures returned from `execution::get_completion_signatures(s, env)` contain three elements:

(3.1) — `execution::set_value(string_view)`
(3.2) — `execution::set_error_t(error_code)`
(3.3) — `execution::set_stopped_t()`

4 The default implementation uses DNS (as specified by a sequence of RFCs) to resolve the address into a name. How the operation is done exactly is implementation specific.

5 [*Editing Note:* There should probably a version of the operation yielding a sequence of results. Most likely the proper approach to do is to use sequence senders which are, however, not yet in a any proposal being discussed. – *end note*]